An Overview of the Java Embedded Micro-kernel Architecture

Hong Ong
Distributed Systems Group
University of Portsmouth

hong.ong@port.ac.uk
http://dsg.port.ac.uk/hong

DSG Seminar Nov. 1st 2002

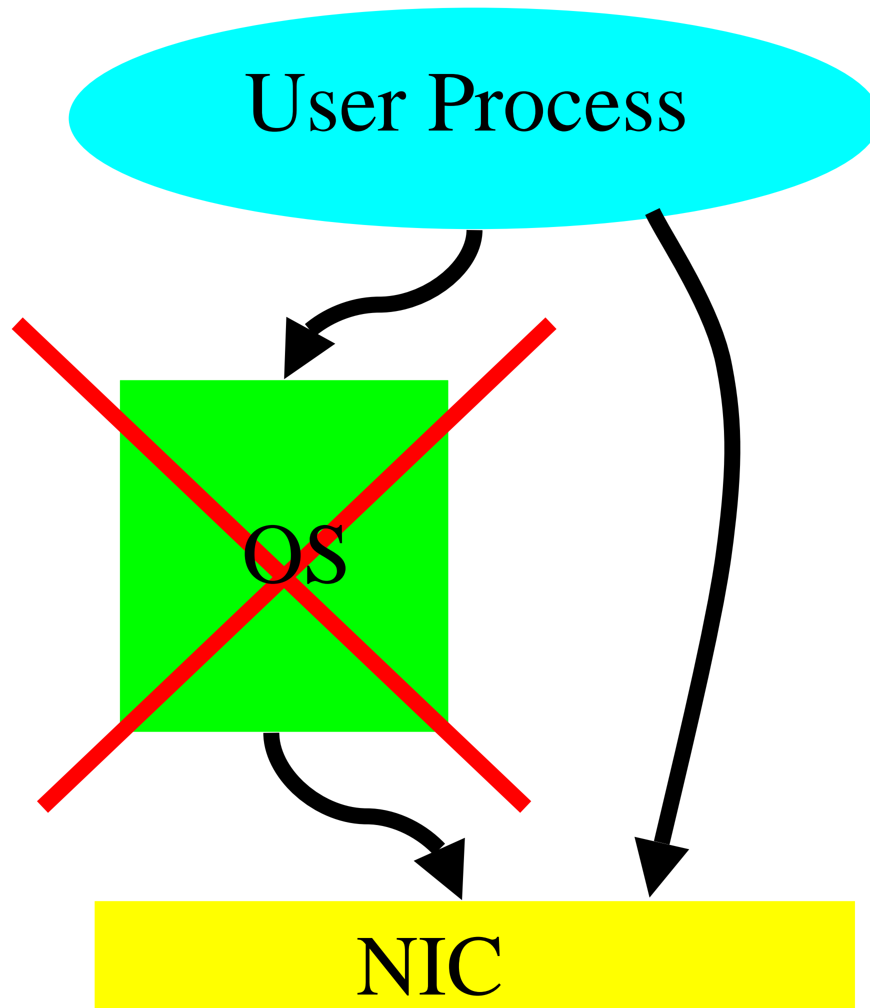
Outline

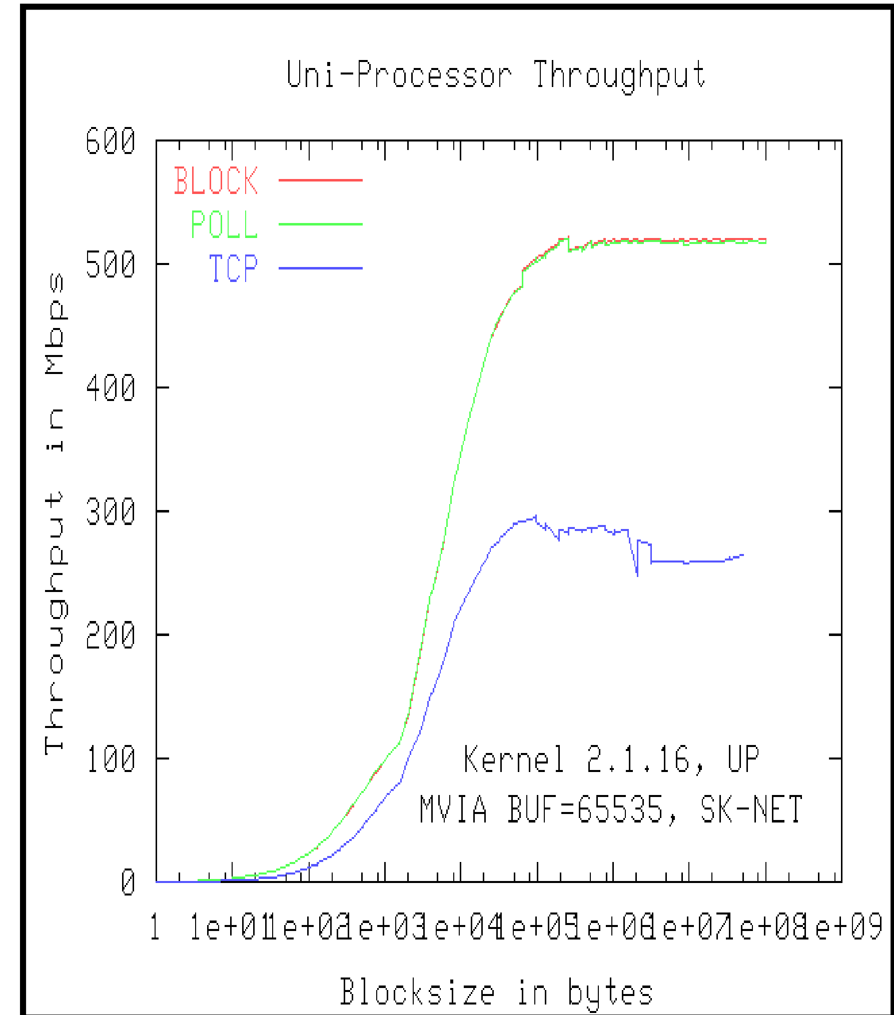
- Project Goals.
- JEM architecture overview.
  - Address space,
  - Threads,
  - IPC.
- Implementation issues.
  - Compilation, memory allocation and mapping, etc.
- Status.

Original Project Goals

- Our original goals were:
  - To design an Abstract Device Interface (ADI) for MPJ.
  - To improve the communications performance of MPJ.
- User-level networking.
  - Virtual Interface Architecture (VIA)
  - InfiniBand
  - Schedule Transfer Protocol (STP)

UNET - General Idea





Things Change ...

- Device has more powerful processors and more memory. For example,
  - Myrinet - 8 Mbytes of RAM, 200Mhz RISC processor.
- A new opportunity:
  - Not just offload protocol processing but also can offload service processing !!
- A new problem:
  - We need a runtime environment to host and manage the service modules (applications).

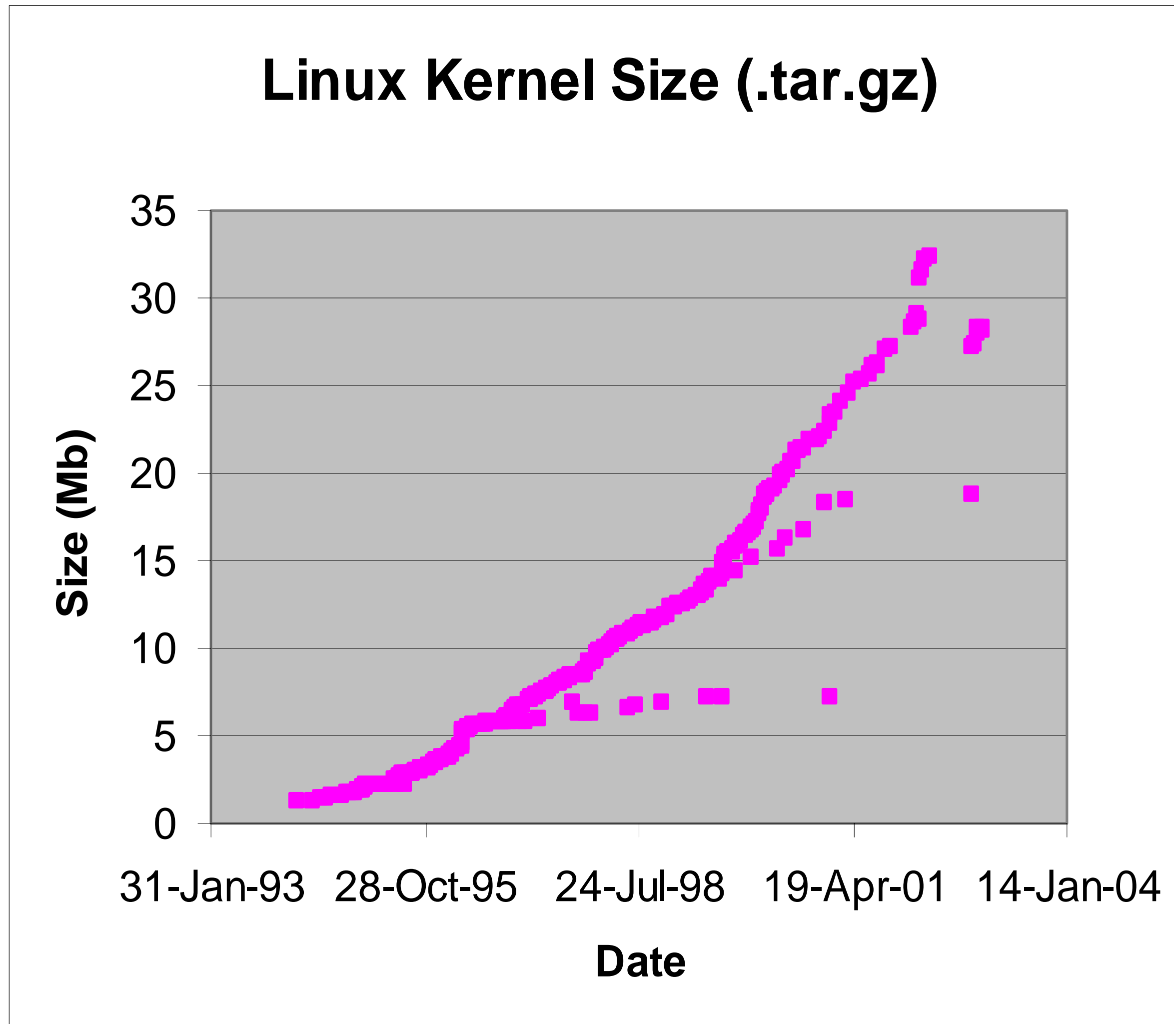
New Project Goals

- A generic runtime environment that can be used in a range of embedded devices:
- A micro-kernel that can provide the necessary mechanisms for the various runtime aspects of the system.
- Service modules that can be dynamically loaded at runtime.
- Micro-kernel + Service modules should be Java-based … well, as far as possible …

Why Java ?

- Classes encourage modularity and data abstraction.
- Safety features (such as array index checking) improve robustness.
- Exceptions make it easier to handle error conditions.
- Many useful packages are available, e.g., networking, GUI, etc.
- Other reasons ...

Why not Linux?



For reference:

- Linux 2.4.18 = 2.7 million lines of code
- uClinux = 512KB size
- JEM = ~12 K lines of codes, expects ~250-300KB (without optimization)

What's out there?

- 1st generation µ-kernels
  - Mach — CMU, OSF
  - Chorus — Inria, Chorus
  - Amoeba — Vrije Universiteit
  - L3 — GMD
- 2nd generation µ-kernels
  - Spin — Washington
  - Exokernel — MIT
  - L4 — GMD / IBM
  - Java OS
  - J-Kernel

External Pager

User-Level Driver

User-Level Address Space

The Great Promise

- Coexistence of different
  - APIs
  - file systems
  - OS personalities
- Flexibility
- Extensibility
- Simplicity
- Maintainability
- Security
- Safety

The Great ~~Promise~~ Disaster

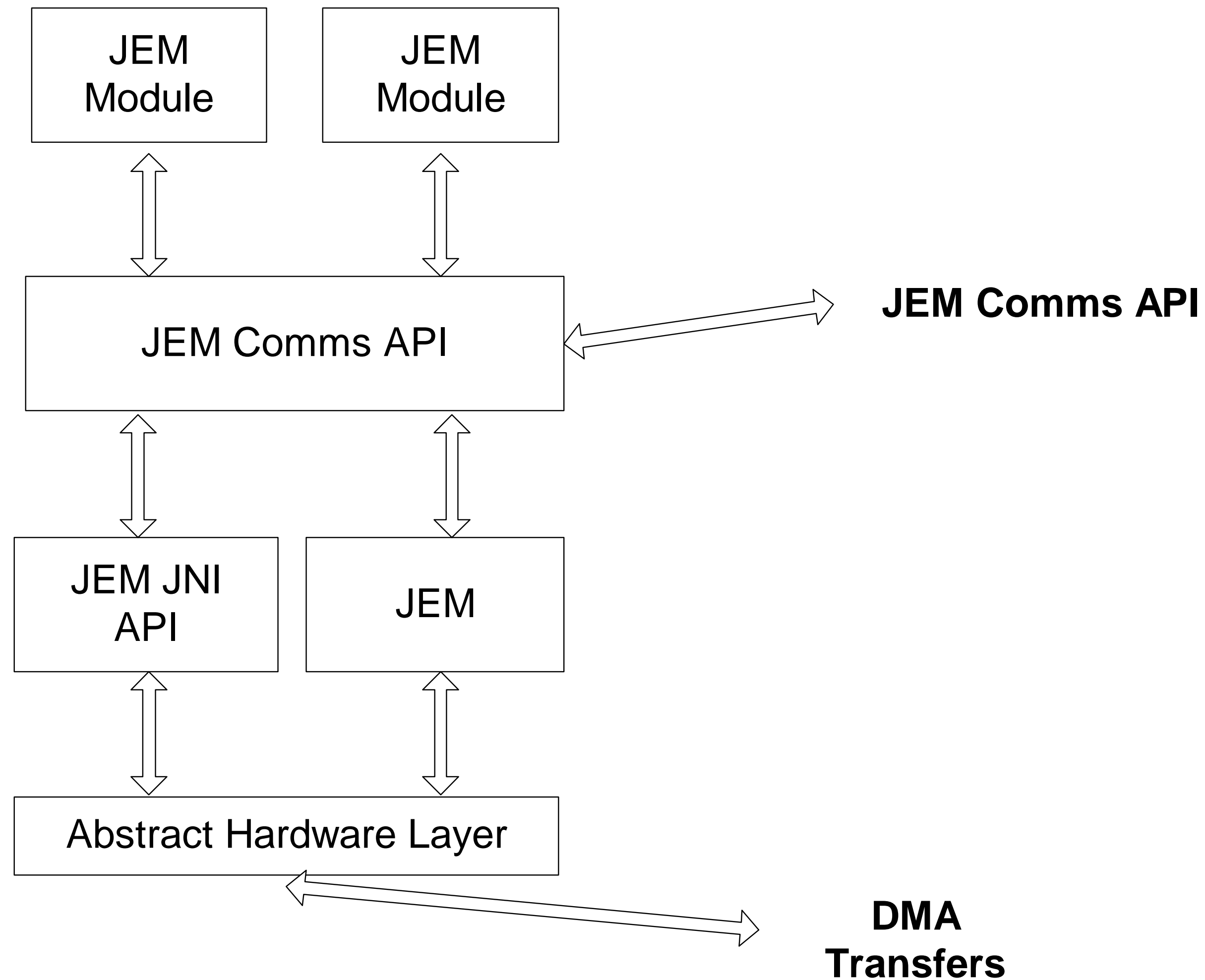
- Coexistence of different
  - APIs
  - file systems
  - OS personalities
- Flexibility
- Extensibility
- Simplicity
- Maintainability
- Security
- Safety

- SLOW
- UNFLEXIBLE
- LARGE

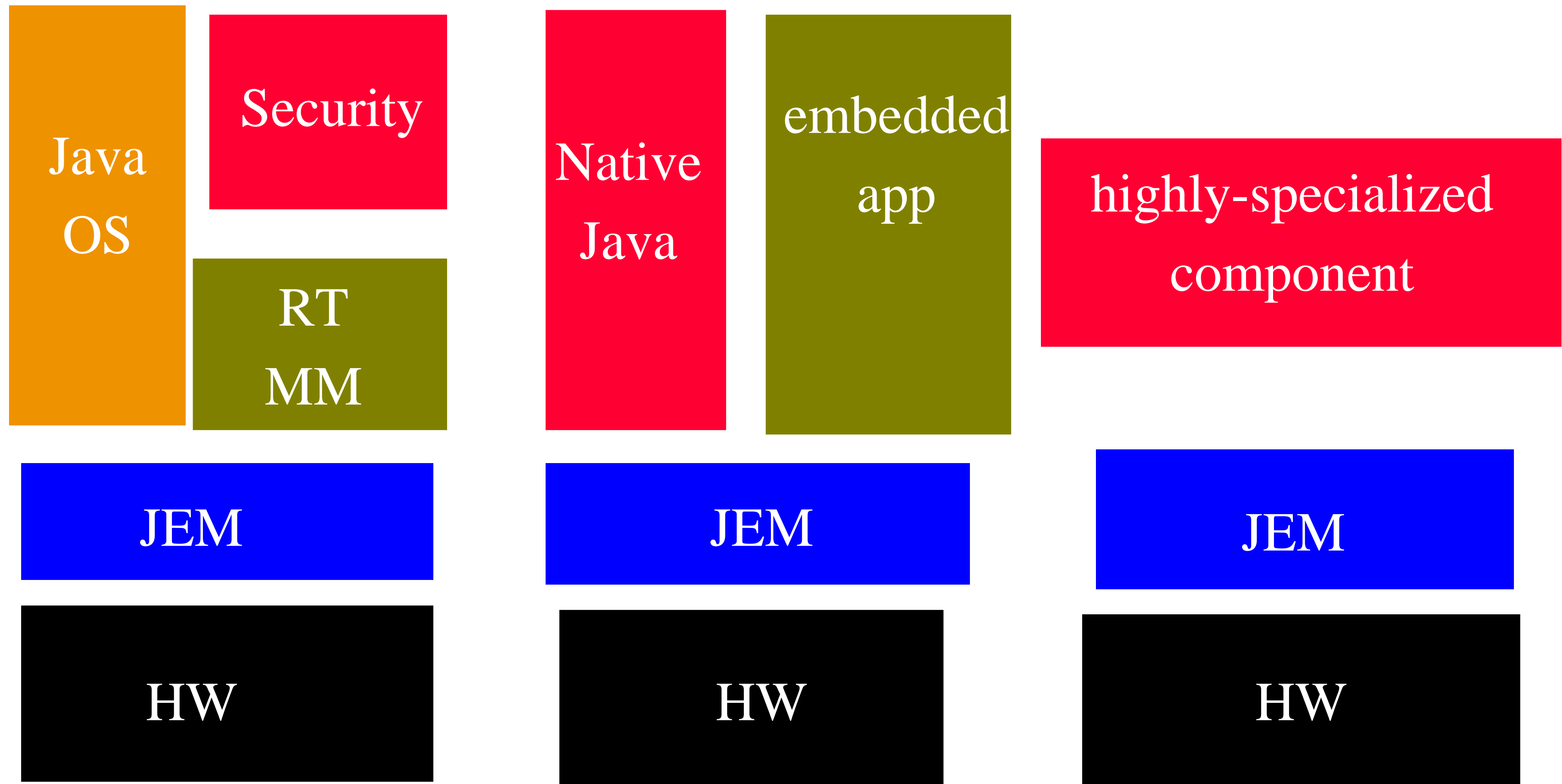
JEM Construction

- Leverage lessons and designs from other projects.
  - Micro kernel – L4, QNX.
  - Safe language environemnt – J-Kernel, SPINE.
  - UNET – Jaguar, PANIC.
- The idea: keep the kernel minimal.
  - all services are implemented outside the kernel as servers that execute in user-mode.
- The problem: security.
  - Not all services can be fully trusted by every user and by every other OS service
- The solution: the address-space paradigm.
  - Any server (and any user) has its own private address space

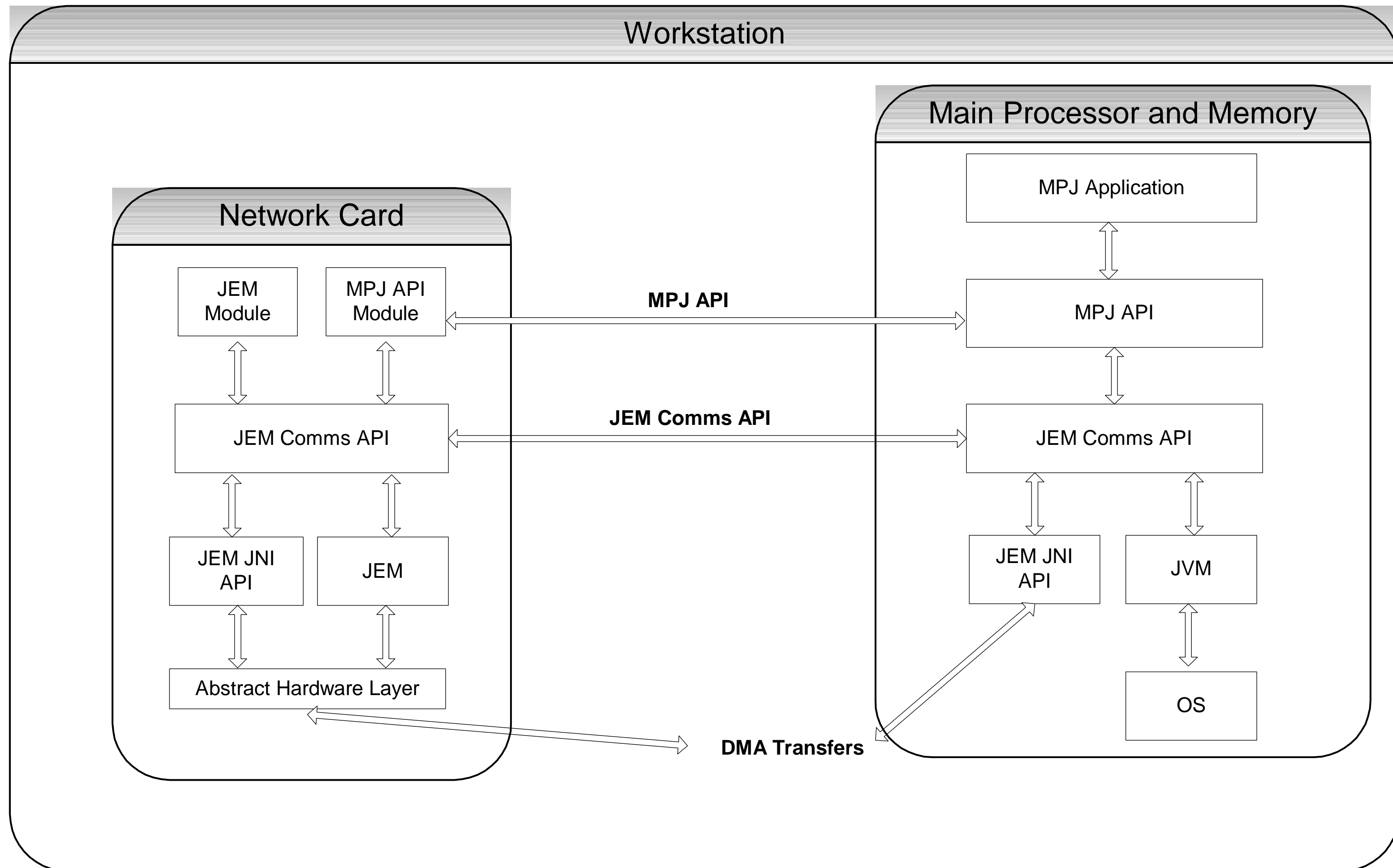
JEM Architecture



Possible JEM Configurations



An Example – Configuration #2



JEM Fundamental Abstraction

- Address Space
- Thread
- IPC

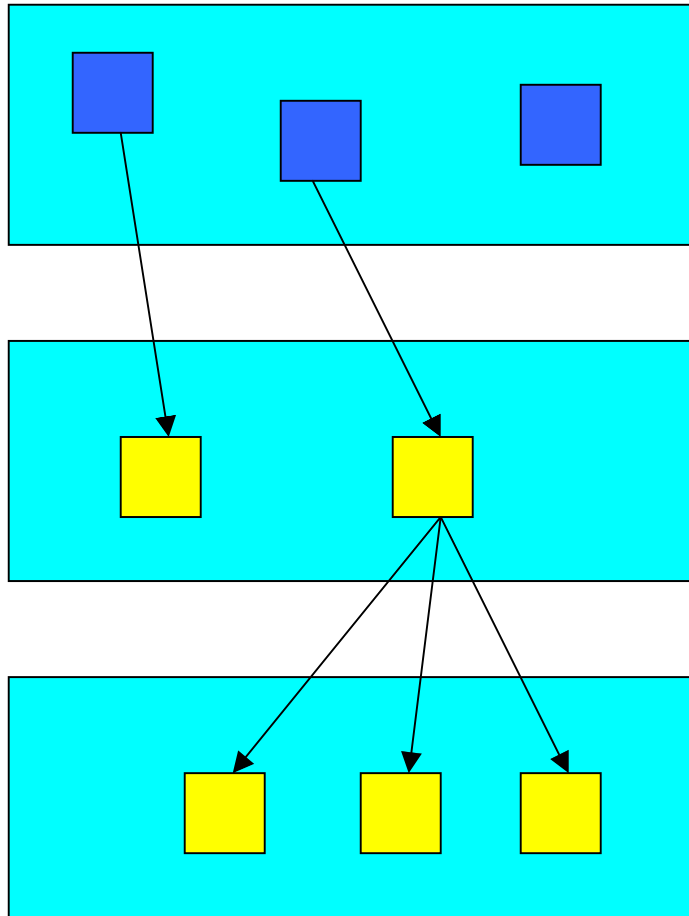
Address Spaces

- At the hardware level, an address space is a mapping which associates each virtual page to a physical page frame or marks it 'non-accessible'.
- The micro-kernel hides the hardware details of address spaces.
- But, allow implementation of arbitrary protection (and non-protection) schemes on top of the micro-kernel.

Recursive Mapping

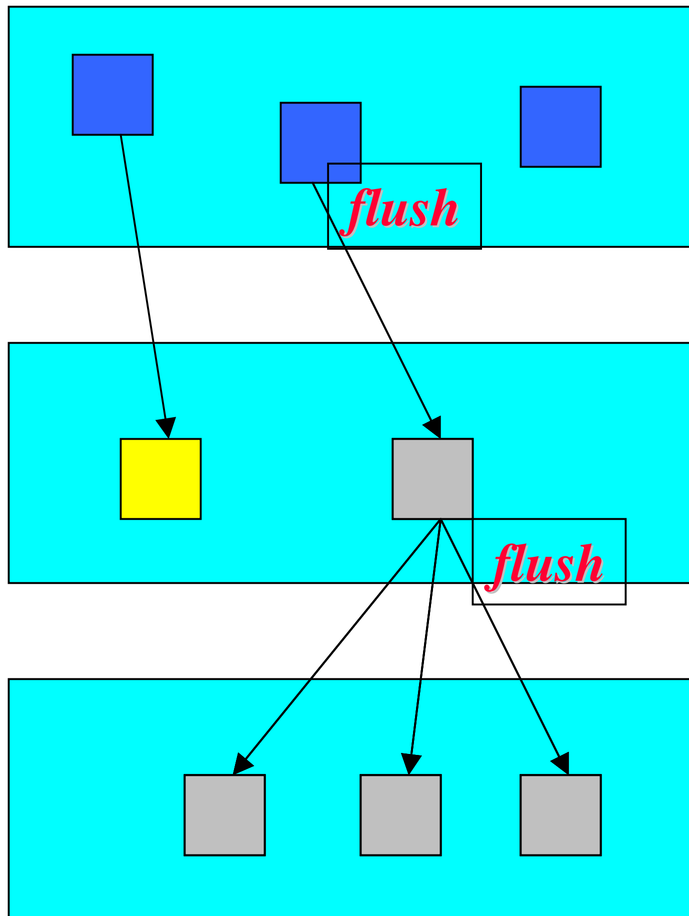
- The basic idea is to support recursive construction of address spaces outside the kernel.
- By magic, there is one address space S0 which essentially represents the physical memory and is controlled by the first subsystem S0.
- At start time all other address spaces are empty.
- For constructing and maintaining further address spaces on top of S0, the micro-kernel provides three primitives:

Address Space Primitive: map()



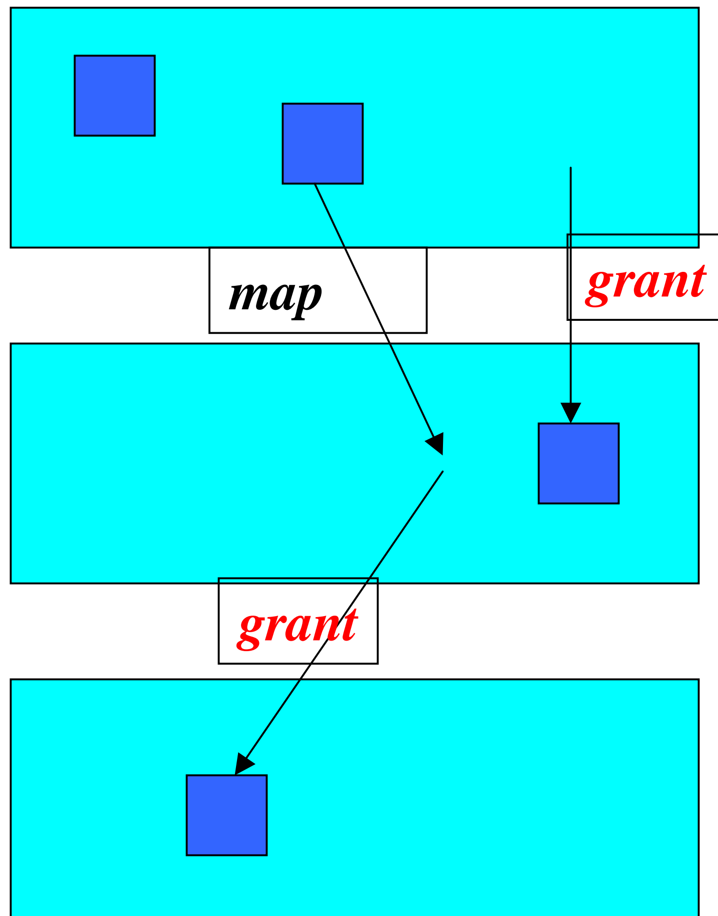
- The owner of an address space can *map* any of its pages into another address space, provided the recipient agrees.
- Afterwards, the page can be accessed in both address spaces.
- The recipient can (recursively) map the page.

Address Space Primitive: flush()



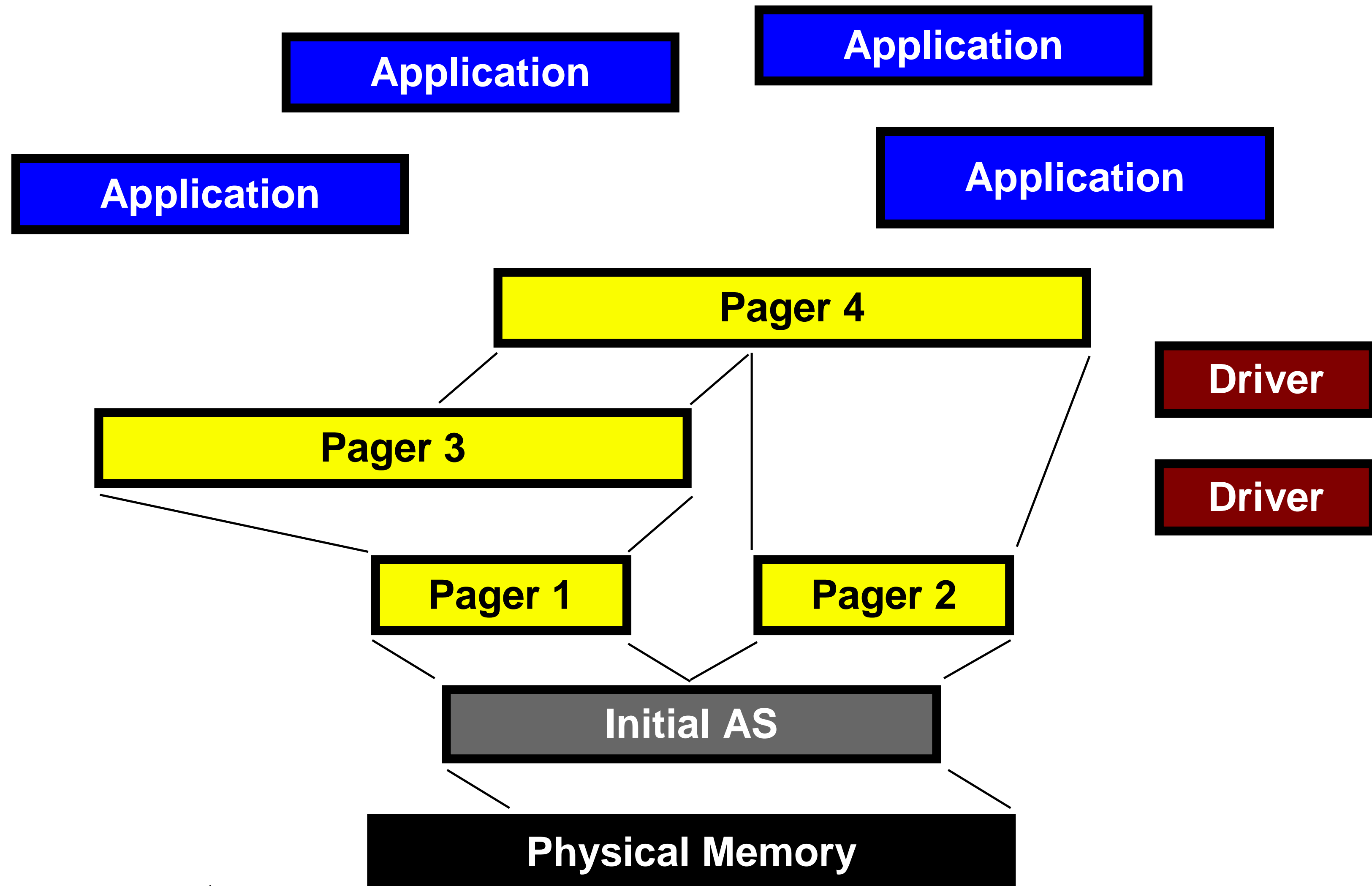
- Owner can *flush* any of its pages.
- The unmapped page remains accessible in the owner's address space, but is removed from all other address spaces.
- The operation is safe since the users of these pages have agreed to accept a potential flushing.

Address Space Primitive: grant()



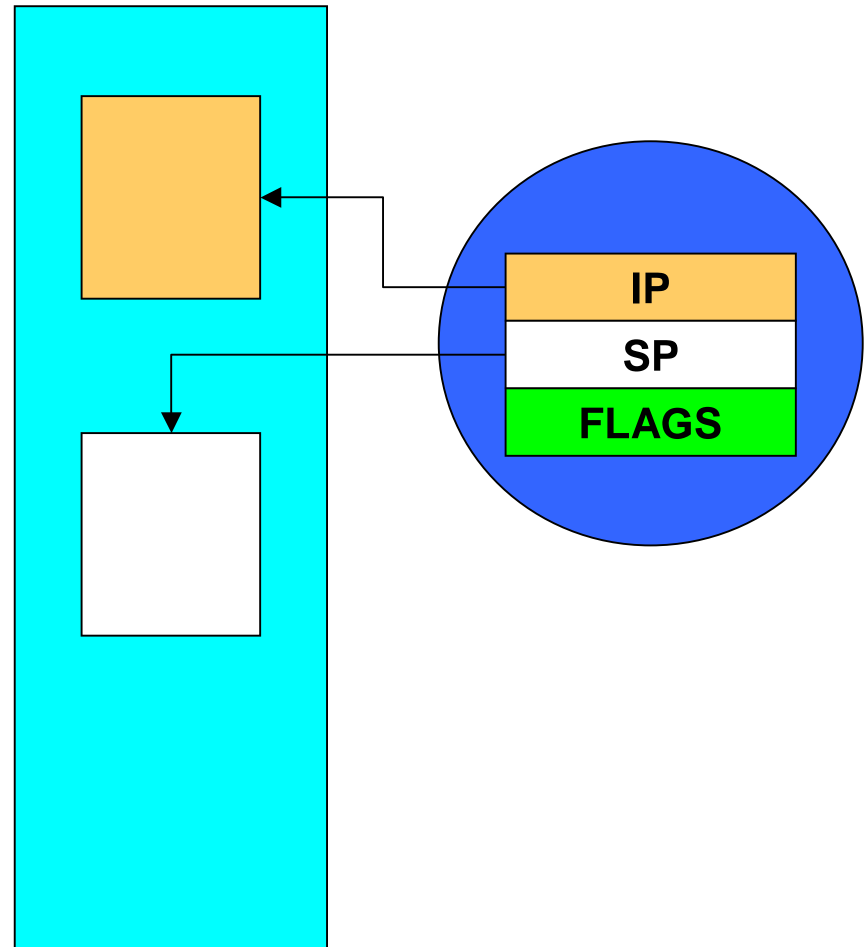
- The owner of an address space can grant any of its pages to another address space
- The granted page is removed from the owner's address space and included into the recipient's address space.
- The important restriction is that the granter can only grant pages that it can access.

Address Spaces: Summary



Thread

- A thread is an independent flow of control inside an address space.
- Internal properties
  - stack
  - status
    - e.g. FLAGs, privilege,
    - OS-specific states (prio, time...)
  - address space
- External properties
  - unique id
  - communication status



IPC

- Operation types:
  - send
  - receive from
  - receive (from anyone)
  - call (send to and receive from callee)
  - send & receive (from anyone)
- Timeouts:
  - snd, rcv, snd pf, rcv pf
  - 0, infinite, 1us ... 19 h (log)
- Message types:
  - register
  - dir string (opt)
    - up to 2M
  - indir strings (opt)
    - up to 15 x 4M
  - map pages (opt)

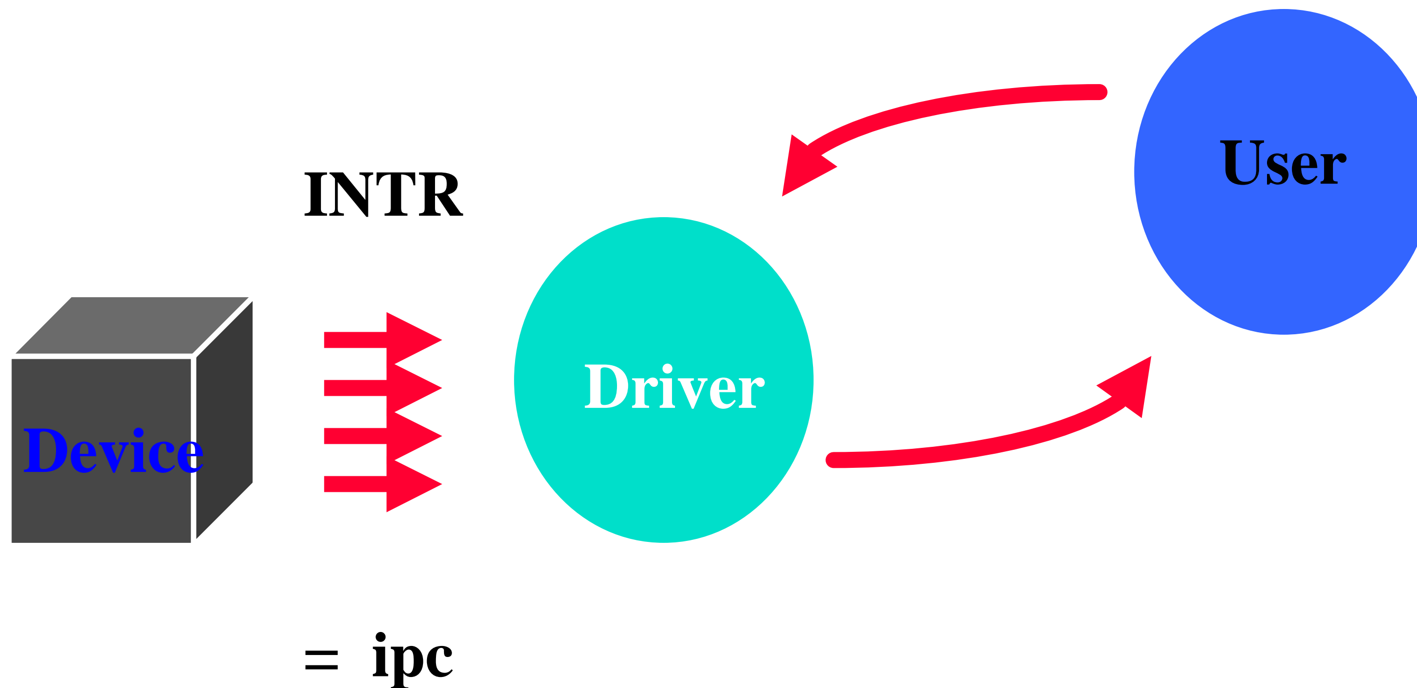
Interrupts

- The natural abstraction for hardware interrupts is the IPC message.
- The hardware interrupt is regarded as a set of threads that send empty messages (only consisting of the sender id) to associated software threads upon device interrupt.
- A receiving thread determine the nature of interruption based on message source id.

I/O

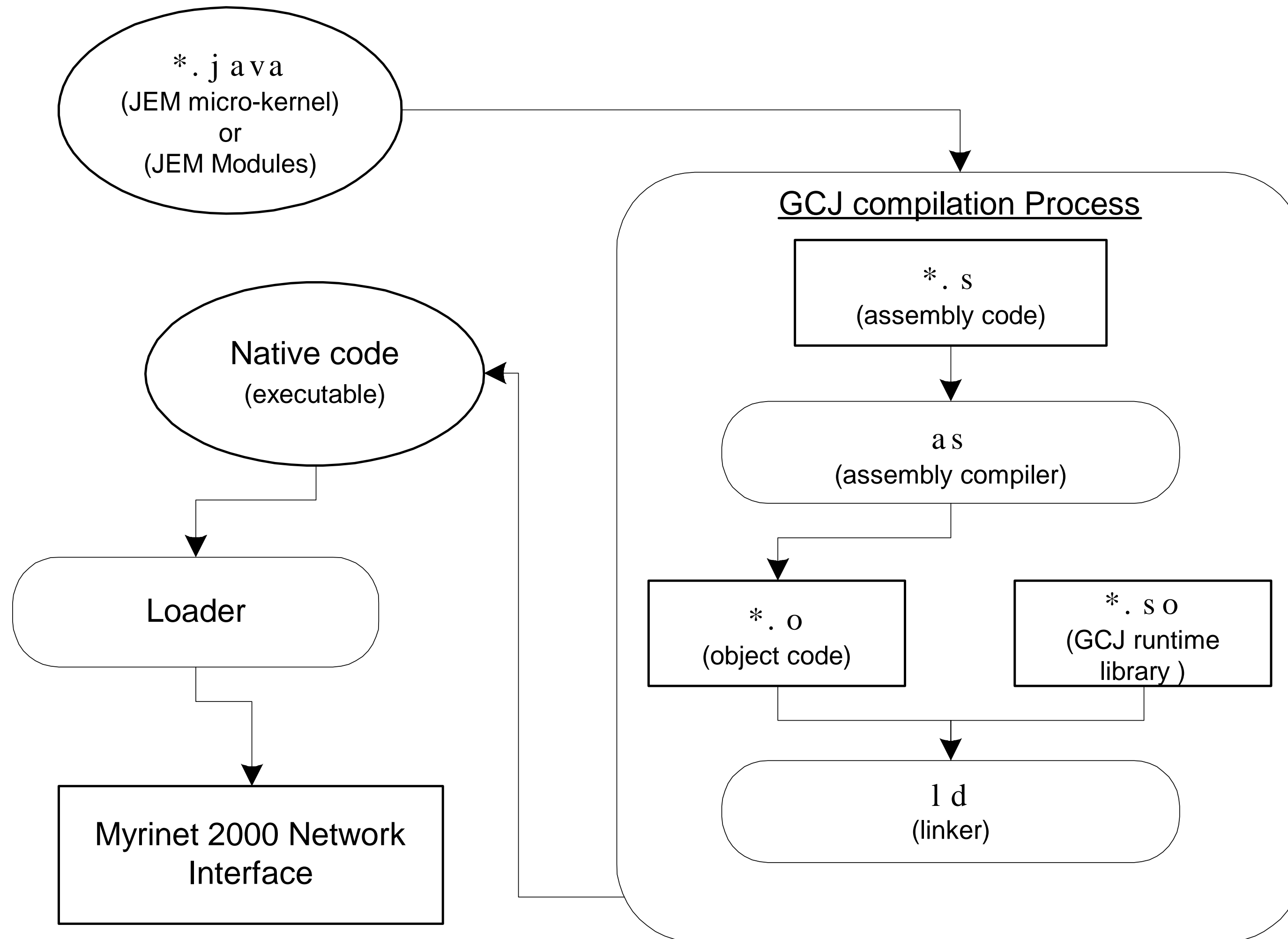
- An address space is the natural abstraction for incorporating device ports.
- This is obvious for memory mapped I/O.
- The granularity of control depends on the given processor.
- Controlling I/O rights and device driver is thus also done by memory pagers and pager on top of the micro-kernel

Drivers at User Level



- **IO ports: part of the user address space**
- **interrupts: messages from hardware**

Implementation Strategy



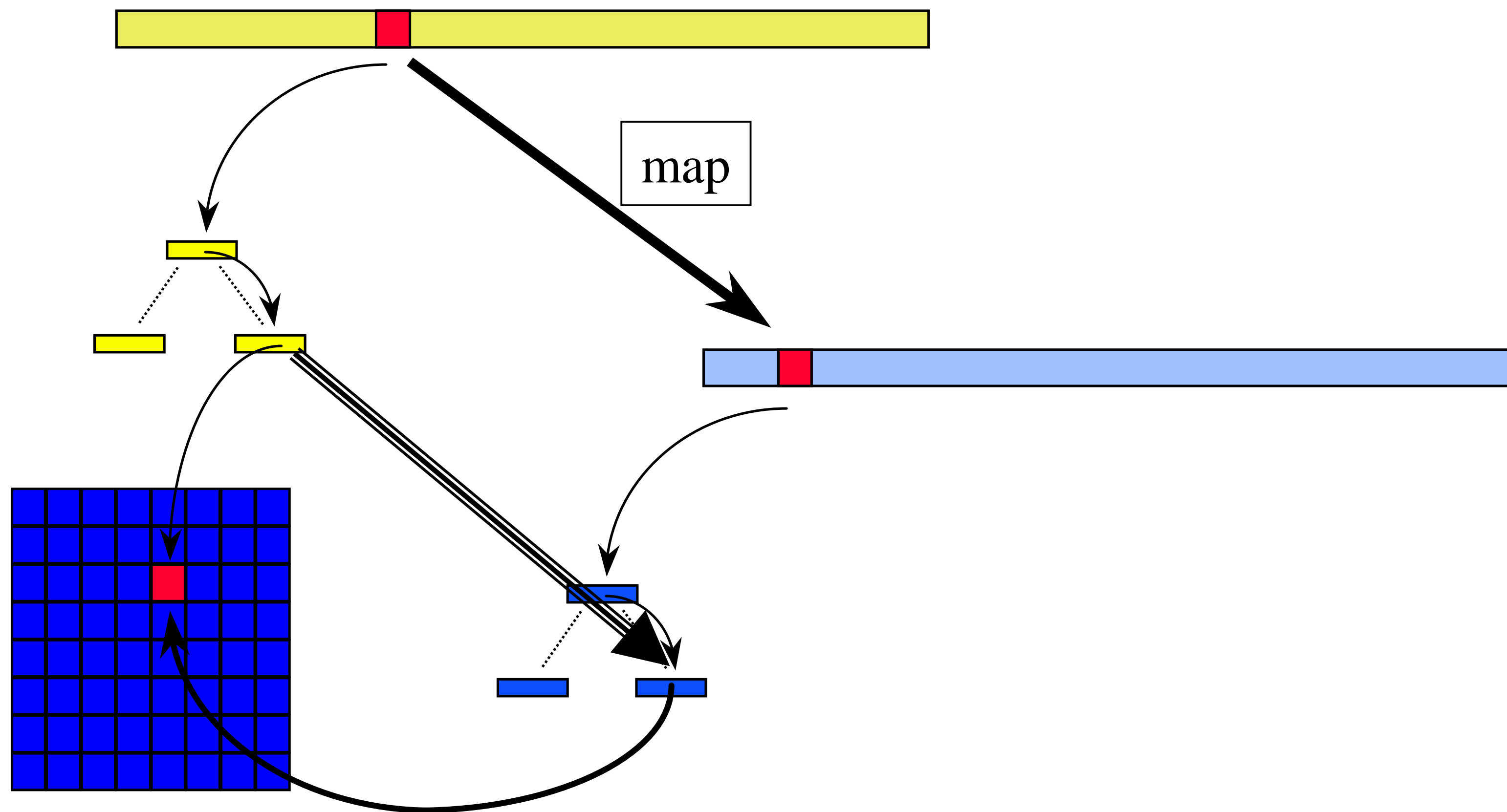
Implementation Issues

- Our Primary Hard Problems:
  - Java memory allocation support – the new operator and finalization (garbage collection),
  - Java thread support,
  - Java Runtime library support.
- Other Hard Problems:
  - Require a robust and general AHL Specification.
  - Performance.

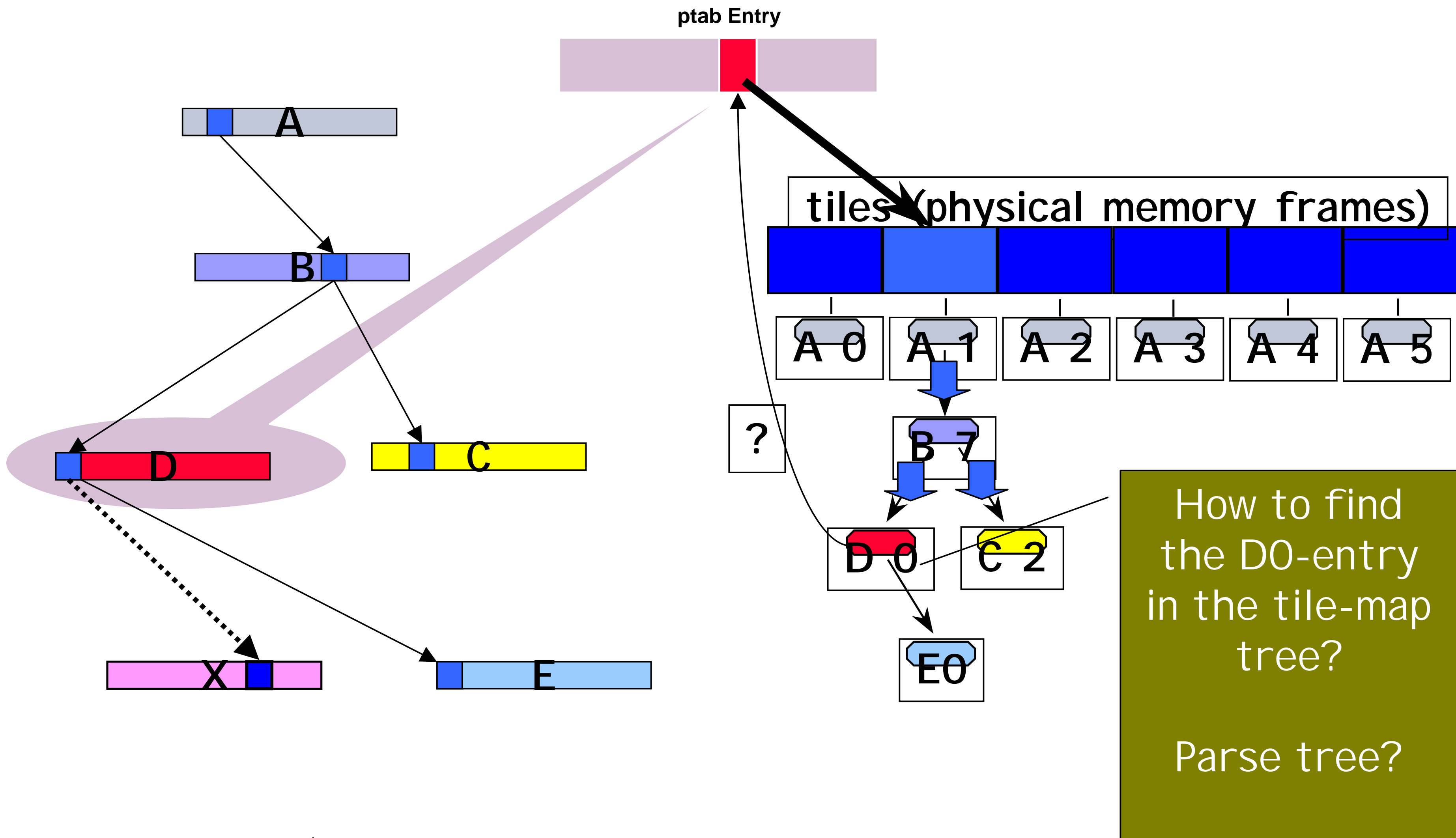
GCJ + libgcj

- Intricate relationship between the GCJ compiler and its runtime support.
- A good thorough knowledge of libgcj internals is required.
- Enable the JEM code to access the meta-information generated by the compiler.
- Wire the GCJ runtime primitives back to the JEM runtime primitives.

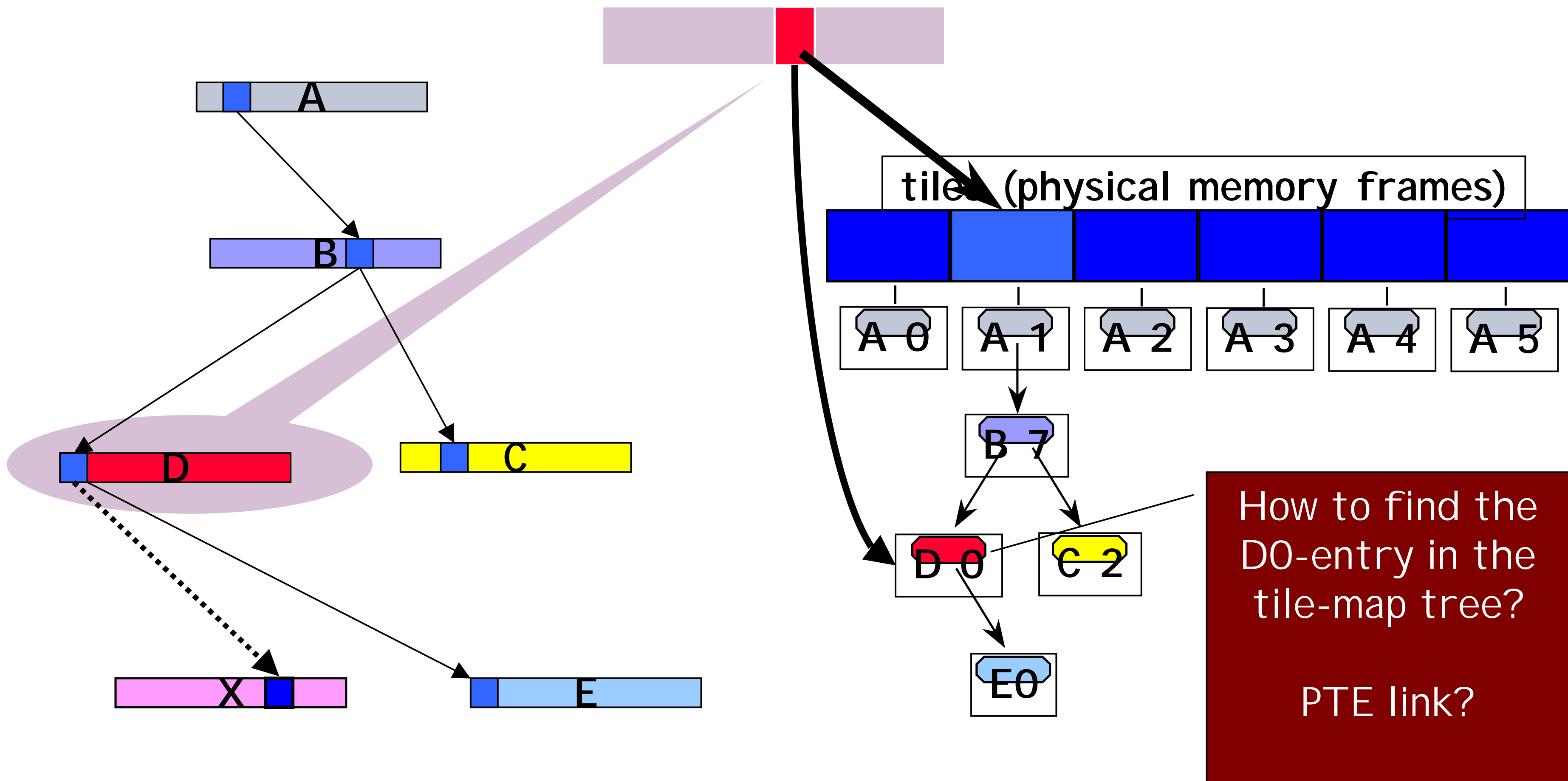
Address Space Mapping



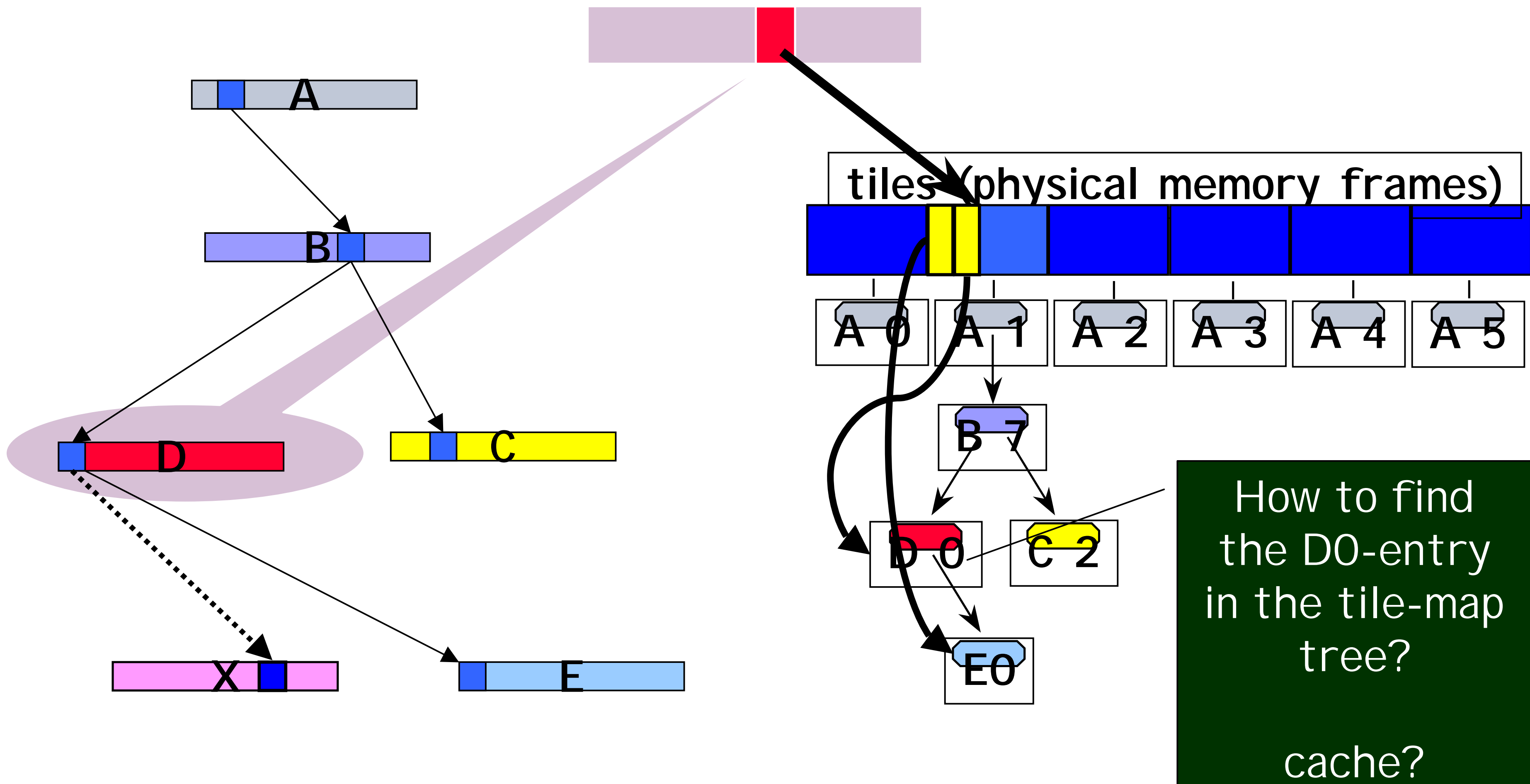
Mapping Database, Tile-map Trees



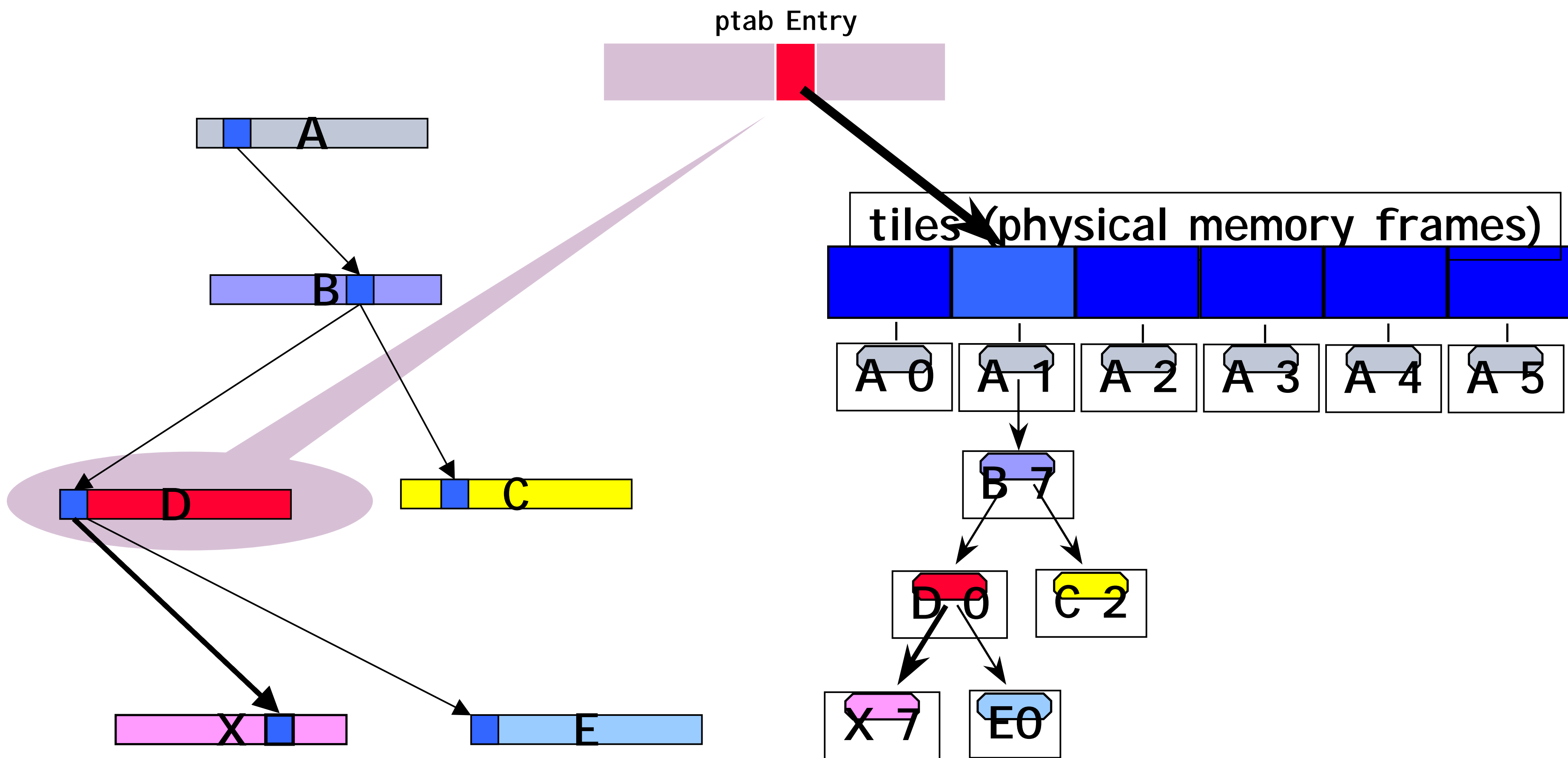
Mapping Database, Tile-map Trees



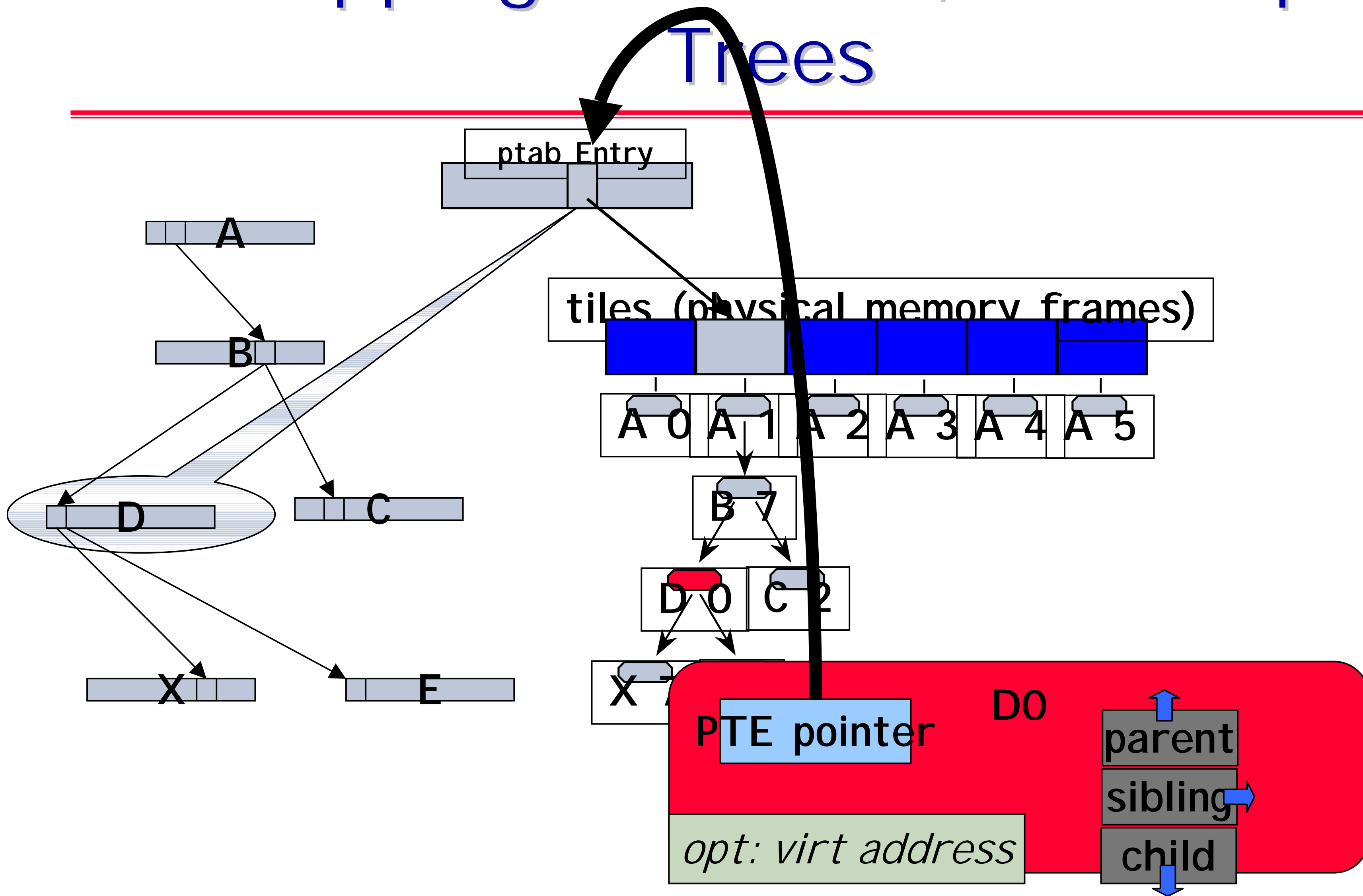
Mapping Database, Tile-map Trees



Mapping Database, Tile-map Trees



Mapping Database, Tile-map Trees



Status

- Done:
  - GCJ + Libgcj modification.
  - Device independent part (almost, need to beautify/tidy the code).
- To Do:
  - Device dependent.
  - AHL Specification.
  - Benchmark – IPC, Memory mapping, etc.
  - Documentation!!